Ensuring Application Integrity in Shared Sensing Environments

Pedro Javier del Cid
iMinds-DistriNet
Katholieke Universiteit Leuven
Leuven, Belgium

Danny Hughes, Sam Michiels, Wouter Joosen
iMinds-DistriNet
Katholieke Universiteit Leuven
Leuven, Belgium

ABSTRACT

Smart Environments, such as smart offices, must support multiple applications that are deployed and managed by different parties. Smart Environments are 'always on' and application software must therefore be deployed, configured and reconfigured while the system is running. Re-configurable component models provide the basic mechanisms necessary to achieve runtime reconfiguration. However, in cases with shared component instances, ensuring application integrity during 3rd party reconfiguration leads to high developer effort and disruption. This paper addresses this problem through *Composition-Swapping*, an extension of re-configurable component models wherein state management delegation and extended component meta-data are used to support component-sharing and ensure application integrity. We demonstrate that Composition-Swapping reduces reconfiguration effort and disruption for four concurrently running applications on a real-world smart office environment.

Categories and Subject Descriptors

C.3 [**Special purpose and application based systems**]: Real time and embedded systems

Keywords

wireless sensor networks, software reconfiguration, sharing

1. INTRODUCTION

Smart environments embed intelligence and enable context-awareness using resource constrained Wireless Sensor and Actuator Networks (WSANs). As multi-purpose sensing infrastructures, emerging smart environments will host a range of applications that may be managed, deployed and used by different parties [15]. As it is critical to reduce deployment and administrative costs of sensing infrastructures [15, 24], smart environments should be designed to maximize the number of concurrent applications they can support. This makes effectively sharing WSAN resources between concurrent applications of paramount importance. Smart environments are 'always-on' and application software must therefore be deployed, configured and reconfigured while the system is running. To address the need for runtime reconfiguration of WSANs, several runtime re-configurable component models have been proposed, such as: RUNES [5], LooCI [11] and Lorien [21]. In addition, lightweight generic re-configurable component models such as OpenCOM [6], Fractal [4] and OSGi [19] have also been used in WSAN scenarios. These models offer the basic mechanisms to achieve runtime reconfiguration, but lack appropriate mechanisms to support shared sensing environments.

Our real-world shared sensing environment, a smart office WSAN, provides a shared and re-usable platform to sense environmental conditions and actuate appliances in our research facility. The core of the smart office is a network of AVR Raven sensor nodes [1], which offer an 8 bit 16MHz microcontroller and 16KB of RAM. To enable sharing in such a resource constrained environment, every processor cycle, byte of memory and radio transmission must be carefully considered. Four distinct sensing applications (elaborated further in Section 2.2) were incrementally deployed on the shared infrastructure. This was a process which led to disruptions and considerable overhead in reconfiguration effort.

One of the main challenges of shared sensing is to allow new applications to be deployed on the infrastructure without compromising existing applications [15, 24]. In our work we focus on the integrity of existing applications as a new application is added to a shared infrastructure. We have analyzed the adverse effects the reconfiguration of an application may have on co-executing applications. Where we quantified the disruption, reconfiguration effort and resource efficiency incurred in the process of ensuring application integrity. We define application integrity as the consistent functioning of an application. Consistent functioning requires that applications execute in accordance with their specification, without faults due to resource competition or disruptions due to software reconfiguration. During reconfiguration we assume no pre-deployment coordination, thus the deploying party has no a-priori knowledge about existing applications and needs to gather information about existing software compositions using reflective mechanisms.

Two primary classes of approaches have been proposed which are designed to allow for the safe co-execution of component based applications in pervasive environments. The first, focuses on offering virtually separated networks leveraging system Virtual Machine (VM) type of concurrency. As in SenShare [15], where they use the capabilities offered in the embedded Linux OS to provide the required isolation between applications. However, this requires of multiple orders of magnitude more resource than available in the nodes we target. The second, are those that leverage code-

CBSE'14, June 30–July 4, 2014, Marcq-en-Baroeul, France.

http://dx.doi.org/10.1145/2602458.2602474 .

recompilation based optimization techniques. As in [12], where at compilation time, application specifications are parsed to find conflicts. However, this entails a disruptive reconfiguration process that requires that all potential configuration conflicts be identified at development time. Furthermore, it requires considerably more resource than available in the nodes we target.

Based on our experiences with our smart office, we have distilled three key requirements for resource sharing in WSAN environments:

1. *Minimize disruption* caused during 3rd party reconfiguration while not restricting component interactions.
2. *Minimize reconfiguration overhead* in terms of developer effort (decisions required, commands issued).
3. *Minimize runtime resource consumption* due to concurrently executing applications (i.e. CPU, RAM, flash memory, # of nodes required) by effectively sharing resources between applications.

To address these requirements, we propose **Composition- Swapping**, wherein a running instance of a software component is shared and reconfigured at runtime to support multiple concurrent applications with different configuration requirements while ensuring application integrity. During Composition-Swapping, isolation between applications is achieved by executing one composition per time period. To support component-sharing and enable Composition-Swapping two key extensions are made to re-configurable component models. First, *state management delegation*, allows our software framework to isolate and manage component configuration and coordination on a per-application basis. Second, *we extended component meta-data*, to describe performance restrictions and application constraints. The former as component annotations and the latter as configuration meta-data. We demonstrate the benefits of Composition-Swapping in a real-world smart office infrastructure, where it (i) eliminates disruption during reconfiguration, (ii) reduces reconfiguration effort, and (iii) makes more efficient use of resources.

2. SHARING IN COMPONENT BASED APPLICATIONS

This section first provides basic insight into Component Based Development (CBD), followed by details on ensuring application integrity and then a summarized description of common reconfiguration challenges. Finally, it highlights the two main causes of this overhead and disruption. CBD is based on the concept of providing generic and reusable software building blocks, i.e. components[1], that can be composed together by third parties into software compositions in order to form distributed applications. The support offered by the component runtime is usually limited to the assembly, introspection (i.e. retrieve information) and reconfiguration of components. Component runtime refers to middleware implemented to provide an execution environment based on a model.

2.1 Ensuring Application integrity

In order to manually ensure the integrity of applications, the deploying party must check three primary issues:

1. There must be no conflicts, between the component configurations of the applications involved, that may lead to faults. Faults are generated when component output does not comply with the expected data accuracy and timeliness. Accuracy refers to the degree in which the reported data corresponds to the observed phenomena. Timeliness refers to the extent in which the age of the reported data is appropriate for the task at hand.
2. Given the combined resource demands and the performance restrictions of the execution environment, applications must execute in accordance with their constraints. Delay is an example of a performance restriction. Delay refers to the time required, by the different parts of the execution environment, to complete data acquisition or processing. The execution environment refers to the software components and hardware elements invoked during the execution of a software composition. Application constraints describe the different factors which may bound resource allocation and component configuration. For example, data accuracy and timeliness.
3. During the reconfiguration process, disruption must not occur. Disruption refers to application down time created by the interruption of its data-flow.

2.2 Reconfiguration challenges

The objective of this subsection is to highlight some reconfiguration challenges commonly encountered in shared WSANs. Challenges that often lead to increased effort, disruption and faults. Before we present theses challenges, we will briefly introduce the four applications deployed in our smart office. As they will aid in the exemplification of these challenges and will also provide the reader with insight into the applications that were used during the evaluation.

2.2.1 Deployed applications

The application deployed were: HVAC, workplace safety, comfort and security. We describe each in the following enumeration:

1. The HVAC application was designed to control ventilation and illumination appliances independently for each desk in our offices. Actuation is controlled based on personalized temperature (temp) and light thresholds. Sensing happens stochastically, i.e. unpredictably, only when the Radio Frequency IDentification (RFID) card of the corresponding employee is detected.
2. The safety application was designed to identify potential health risks based upon the filtered and correlated readings from temp, Carbon Monoxide (CO), methane (CH4) and humidity (RH) sensors (see Fig. 1A). Sensing in this application is periodic and happens every 7 minutes when a timer expires.
3. The comfort application was designed to determine office comfort levels based on averaged readings from temp, light, CO, CH4, RH and sound (dB) in each office. Application for which sensing should happen every 4 minutes.
4. Finally, the security application is designed to provide access control and motion detection, based on RFID and Passive InfraRed (PIR) motion detection. Application for which motion detection is only activated if no employees are in the office.

2.2.2 Reconfiguration challenges

During the deployment of a new application onto a shared WSAN, there are some challenges that are particularly notable. Primarily because of the effort, disruption or faults they commonly create. In the following enumeration we describe these challenges:

1. *Identifying potential configuration conflicts under the coexistence of periodic and stochastic interactions*. This co-existence

[1]Component refers to the reusable code base, while component instance refers to the runtime entity instantiated from the code base.

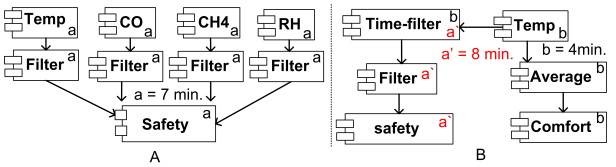

Figure 1: (A) Safety app. (B) Disruption caused during reconfiguration.

creates a degree of uncertainty in the verification of integrity. This is the case because the deploying party is not able to identify nor resolve potential resource contention, as it is unknown when and if concurrent invocation will happen. Thus leaving behind the potential for faults and disruption. For example: in the case of the temperature sensing components for the HVAC and safety applications.

2. *Modifying any running component instance.* Modification made to instances being used, affects all compositions connected to them. An action that potentially leads to disruption and faults. More so if the strategy of filter insertion is used to reconcile different configuration requirements. For example: in the case of our safety and comfort applications, which share temp, CO, CH4 and RH sensors. Reconciliation is needed because one app. requires 4 min. sampling intervals and the other 7 min.. Thus, filter insertion is used to down sample readings and allow components to be shared between applications (see Fig. 1B). Disruption is unavoidably caused during the insertion of filters into the composition and the loss of sensing precision is caused to the safety application as the new sampling interval is 8 min..
3. *Establishing if the planned modifications will adversely affect the integrity of existing applications*. There is usually no explicit and machine parse-able information pertaining to each application's constraints. Therefore, detailed software specifications and in some cases even inter-party coordination is needed to realize this task.
4. *Establishing if the WSAN will support the combined resource demand*. There is usually no explicit and machine parse-able information pertaining to the performance of both: the applications involved, and of the execution environment. Therefore, detailed hardware specification and in some cases even the component's source code is needed to complete this task.

2.3 Causes for the reconfiguration overhead

We have described the need to manually ensure application integrity in order to be able to deploy new applications on a shared WSAN without compromising existing ones. We have also provided a summarized description of challenges that commonly arise during reconfiguration; challenges which often lead to additional reconfiguration overhead, disruption and faults. To conclude we enumerate the two primary causes of the reconfiguration effort, faults and disruption illustrated:

1. Due to a lack of per-application isolation of component configuration and coordination information, any change in configuration or coordination on a shared component instance affects all compositions using that instance. It is thus necessary to *manually* ensure application integrity.
2. Due to a lack of application constraint and performance metadata, additional documentation (e.g. hardware and software specifications) and in some cases, inter-party coordination is needed in order to ensure integrity.

3. RELATED WORK

This section first provides an overview of alternatives for sharing a WSAN. Second, reviews component models applicable in WSANs, and finally, discusses ensuring application integrity for component based systems.

3.1 Sharing a WSAN

Research approaches that allow the sharing of sensing infrastructures can be broadly classified by the sharing strategy they utilize. These are: network partitioning, virtual WSANs, data-centric and node-level. In network partitioning, a part of the network is assigned to each application, e.g. TinyCubus [17]. Virtual WSANs offer virtually separated networks on top of shared nodes (using system VM type support for concurrency), e.g. SenShare [15]. Data-centric approaches present the WSAN as a database, e.g. TinydB [16]. Node-based approaches leverage node-level concurrency models, as those offered by threading libraries or process VMs, e.g. [2, 12, 13]. These broad categories vary primarily in the resource efficiency and isolation they provide. Network partitioning requires node redundancy therefore it is not adequate for smart environments. Virtual WSANs offer a high degree of isolation and allow node sharing but the per node resources required are considerably above those offered by embedded nodes (e..g [15] requires Linux based nodes with 32MB RAM whereas embedded nodes commonly have at most 16KB). Data-centric approaches implicitly support multi-application scenarios, but they do not support the dynamics of smart environments [15, 24].

Node-level approaches can be further classified into coarse-grained and component based (CBD) approaches. Coarse-grained approaches as Umade [2] support multiple applications by implementing each application separately as a dedicated coarse grained module. This leads to a high degree of redundant functionality, which is resource inefficient (e.g. [2] with 7 KB of RAM runs on average only 2 apps. per node). Furthermore, they do not ensure application integrity. Approaches based on CBD, as GatorTech [10], due to componentization, offer the possibility of a higher degree of reuse. However these offer very limited support for isolation, lack the meta-data required to ensure integrity and are resource inefficient (due to redundant component instances). Note: these are all limitations intrinsic to the underlying component model. In the pervasive space, stream processing frameworks [13] and code-based optimization [12], have leveraged component sharing to improve resource efficiency. However, they do not offer the required isolation or meta-data. Furthermore, they require over 10 times more resources than available in embedded nodes and only support periodic interactions.

3.2 Re-configurable component models applicable in WSAN

Existing re-configurable models [4–6,11,19,21] have three major shortcomings, which we elaborate on below.

3.2.1 Resource inefficiency

Stems from the need to rely on multiple instances of functionally equivalent and thus redundant component instances to support concurrent applications. In contemporary models [4–6,11,19,21] it is assumed that each application will be assembled from dedicated instances. An assumption which is not appropriate in embedded WSANs for of two reasons: First, sharing of sensor hardware is a necessity. As it is cost prohibitive and technically unfeasible to have dedicated sensors for each application [13, 24]. Second, improved resource control is obtained by mediating concurrent access to a shared resource using a *Singleton* instance [9]. Furthermore, in

these models, it is assumed that sensor contention will be properly managed by the OS. Which is not the case, as WSAN OSes [22] offer no support to resolve contention over sensors. Instead, these only provide support to read values at each analog or digital IO port [1].

3.2.2 Lack of isolation

In current models [4–6, 11, 19, 21] configuration is part of the internal state of each component instance (in form of parameters) and coordination (i.e. interconnections and interactions) is part of the state managed by the component runtime. However, component configuration and coordination information is not isolated in a per-application basis, thus any change will affect all connected compositions. These models are not concerned with per-application isolation because they do not foresee the need to support component-sharing, which as elaborated in Sec. 3.2.1 is not an appropriate assumption. Fractal [4] and Ernie [20] provide *limited* component-sharing support, as they only allow the sharing of single-configuration components. This is useful to keep state consistent in cases of concurrent access to actuation components (e.g. controlling a fan). However, it does not improve reconfiguration or resource efficiency.

3.2.3 Lack of application constraint and performance restriction meta-data.

In these model [4–6, 11, 19, 21], application constraints are only implicitly represented through the values assigned to component configuration parameters and performance restrictions are not represented at all. Therefore, application developers need access to hardware and software specifications, and in some cases even inter-party coordination is needed to ensure application integrity.

3.3 Ensuring application integrity

In mainstream computing, similar integrity requirements, are present in the dynamic change management, e.g. [14] and adaptive software domains, e.g. [25]. However, there are two key differences between our perspective on integrity and the perspective these approaches commonly adopt. The first, in our work we focus on the integrity of existing applications as new applications are added to a shared infrastructure. Therefore, we consider the adverse effects the reconfiguration of an application may have on co-executing applications. Existing approaches commonly focus more on the safety of transitions as an application moves from one steady-state to another [25]. Therefore, the focus lies solely on a single application and its adaptations. Second, in our work we consider resource constraints and the issues of resource competition, and existing approaches generally do not [14, 25]. Component frameworks, e.g. [6], ease configuration effort due to the reuse of known configurations, but do not ensure application integrity. In embedded WSANs a simplified approach is used where only structural integrity is verified, e.g. [18, 21]. This generally only involves a light-weight mechanism to verify that mandatory component interconnections are satisfied before a component can be activated.

4. RESOURCE SHARING FRAMEWORK

The Resource Sharing Framework (*RSF*) extends re-configurable component models in order to support concurrently running applications in shared sensing environments through a process we refer to as **Composition-Swapping**. Composition-Swapping, allows application developers to deploy applications which share component instances with existing applications, but treat them as if they are dedicated instances. Therefore, developers are relieved from the complexity of ensuring application integrity. As discussed in Section 2, with existing models any modification to the configuration or coordination of a shared component instance affects all associated compositions (see Fig 1B). On the other hand, now consider Fig. 2A and B, where due to their per-application isolation, the temp component can have a unique configuration for each composition without affecting other applications. This isolation is achieved by executing one composition per time-period[2] and swapping composition configurations in and out as required, hence Composition-Swapping. To achieve this, each time a composition is triggered for execution, the RSF configures the corresponding component instances and prepares the composition's structure. In order to obtain higher resource efficiency gains, the RSF supports the sharing of sensing and data-processing component instances.

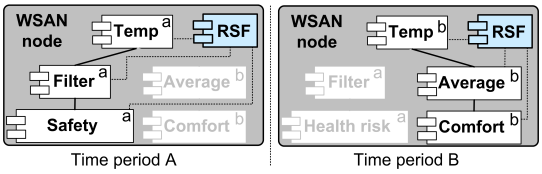

Figure 2: Per-application isolation is achieved by Composition-Swapping.

To elaborate on Composition-Swapping, we discuss its involvement in three phases of the software life cycle:

(A) *Development time:* The component developer follows a patterned component implementation to enforce state management delegation and annotates components with performance meta-data.

(B) *Configuration time:* The application developer makes application constraints explicit and assembles the application using the standard mechanisms offered by the component runtime.

(C) *Runtime:* The RSF realizes Composition-Swapping and ensures application integrity.

We delve further into each of these in Sections 4.1 to 4.3 respectively and end with a note on modifiability in Sec. 4.4.

4.1 Development time

The RSF is designed as a modular add-on for existing component runtimes and is itself implemented as a standard component. It is not a programming approach, therefore, applications are still designed according to the chosen component model. However, there are two core extensions needed in the implementation of each shared component: State management delegation and the annotation of component meta-data.

4.1.1 State management delegation

State management delegation allows the RSF to isolate and manage component configuration and coordination on a per-application basis. This idea is inspired by state management in multi-tenant middleware, e.g. [23]. State management delegation is achieved by enforcing a patterned component implementation, as shown in Listing 1. Each component instance interacts with the RSF through `Idelegate`and `Iexecute`.`Idelegate` allows component instances to delegate their state to the RSF. In turn, `Iexecute` allows the RSF to return their state and control their execution. To achieve delegation, the developer must implement three things:
(i) A call to the `delegateToRSF()` method (Line 9) from within

[2]A notion similar to time-slotting, commonly used in OSes [22] to support the execution of concurrent threads.

the `setProperties()` method. In this way, every time the application developer submits application data through the `setProperties()` interface (Line 6), it will be immediately delegated to the RSF.
(ii) The `submitContext()` method (Line 10), which handles context occurrences (e.g. detection of an RFID card). For instance in cases where a component should sense when an RFID card is detected. It must then be delegated with the `delegateContextToRSF()` call in Line 11.
(iii) The `execute()` method (Line 12) contains the component's functional code, its configuration and control. In the case of sensing components, a `copy()` method must also be provided, which allows the RSF to request a copy of the latest reading.

```
1  componentType = type; // e.g. sensing, data processing
2  mutexIds; // IDs for applicable mutex locks
3  outputType;// to select algorithm to check accuracy
4  performanceDelay1 = time1;//device specific
5  performanceDelay2 = time2;//peripherals
6  setProperties(configuration){
7      // Delegation call for submitted configuration
8      // and the annotated meta-data.
9      delegateToRSF(configuration,componentMetaData);}
10 submitContext(occurrence){
11     delegateContextToRSF(occurrence);}
12 execute(parameters){
13     // Configure how functionality should execute.
14     // Execute functionality.}
15 copy(parameters){ }
```

Listing 1: RSF-shared component implementation template

4.1.2 Component meta-data

The component developer must specify the component type, applicable mutex locks, output type and the corresponding delays, i.e. performance restrictions. The component type identifies the type of functionality implemented by the component (Line 1 in Listing 1), e.g. sensing, data processing. Mutex locks mediate access to contentious resources which are not externalized through component instances. For example, as there is only one ADC, it must be shared by all analog sensors. At runtime, the RSF locks and releases them when the execution of the corresponding component terminates, to ensure consistent sharing. The component developer must identify where locks are necessary from hardware specifications. The ID of each applicable lock is included in an array (Line 2). She is responsible for using these IDs consistently, i.e. in all components, the same ID refers to the same resource.

Output type is used to select the appropriate algorithm to verify data accuracy and it specifies the characteristics of the data output for each component. For example, temp, light, RH sensors generally have noiseless measurements which are steady, thus are type 1, and use a simple exponential smoothing function for accuracy [3]. For noisy moving values, as those expected from proximity sensors, a type 2 would be used, which uses a double exponential smoothing function [3].

Performance meta-data in the form of delay, allows the RSF to account for the time it takes, each of the involved parts of the execution environment, to complete data acquisition or processing. The component developer, must, through the use of technical hardware specifications ascertain applicable delays caused by hardware elements and measure component execution time to quantify software delays. These measurements must cover the total scope of initialization and computation of each component. For example, analog sensing components must account not only for the initialization of the sensing hardware but also for peripheral resources used. Peripheral resource refer to the secondary resources that are used during the invocation of a sensor (e.g. the ADC circuitry). These delays are measured in time and normally in the order of milliseconds. However, radical variations exist as, chemical sensors with a 24hr. heatup time (due to chemical substrate) and PIR sensors with a 2 sec. capture time. See a sample implementation of a Temp sensing component in Listing 2.

```
1  int cT = 1; // 1= sensing
2  int [] mIds = {3}; // 3 = mutex for ADC
3  int outputType = 1 // 1 = steady noiseless measurement
4  int R_device = 20;//millisec. sensing time
5  int R_peripheral = 10;//millisec. ADC reset-time
6  int copy;
7  void setProperties(char[] configuration){
8      delegateToRSF(configuration,R_device,R_peripheral,cT,mIds);}
9  void submitContext(char[] occurrence){
10     delegateContextToRSF(occurrence);}
11 int execute(){
12         int value = ReadADC(TEMP_ADC_CHANNEL);
13         copy = value;
14         return value }
15 int copy(){
16         return copy;}
```

Listing 2: Example implementation of shared temp component

4.1.3 The differences to non-shared component implementation

The standard (i.e. non-shared) implementation lacks performance annotations and does not delegate its configuration, control or context occurrences. Therefore, it is responsible for persisting its own configuration (Line 3 in Listing 3) and controlling its own execution (Line 4).

```
1  char[] configProperties;
2  void setProperties(char[] configurationProperties){
3      // persist config. properties
4  main control loop {
5      // Implement occurrence handlers and timers.
6      // configure functionality from persisted state.
7      // Execute functionality.}
```

Listing 3: non-shared component implementation template

4.2 Configuration time

At configuration time, the application developer is responsible for two things. The first, quantifying application constraints, and the second, the assembly of the application.

4.2.1 Application constraint meta-data

The application developer must, based on application requirements, identify and quantify applicable application constraints. This meta-data is added to the usual component configuration information submitted during application assembly. Application constraint meta-data allows developers to specify additional constraints that inform the RSF's allocation and verification processes. In this way the RSF is able to ascertain the validity of a given configuration under current system conditions. In our smart office we accounted for accuracy and timeliness. Data accuracy is considered in two ways: The first is designed to determine the 'safe' or appropriate time for a component to acquire or process data. This procedure, which is enabled by default for all components, verifies that component computation is done in accordance with the limitations of the execution environment. The second, is designed to account for faulty or noisy sensors. This is an optional verification procedure which can be activated per application. In our current implementation, this verification is done using either simple or double exponential smoothing [3]. For which, the developer needs to submit the alpha and delta to be used. Alpha indicates how responsive to change the

verification function should be ($0 \geq \alpha \leq 1$). Delta is a percentage which indicates how close an output value must be to the expected value, in order for it to be considered accurate (e.g. safety has `ApC_3 = [`α `=.5,`δ `= 10%])`. Timeliness, is expressed either with the time-drift (e.g. comfort has `ApC_1 = 5%`) or response-time constraint (e.g. HVAC has `ApC_2 = 3 minutes`).

4.2.2 *Application assembly*

The application developer must submit two commands per component instance: First, the configuration of each instance is sent directly to the instance using its `setProperties(configuration)` interface (see Fig. 3B step 1). This configuration data is then automatically delegated to the RSF (step 2). Second, coordination information (i.e. composition structure) for each instance is submitted to the RSF directly using its `setProperties(coordination)` interface (step 3). We elaborate further on each step:

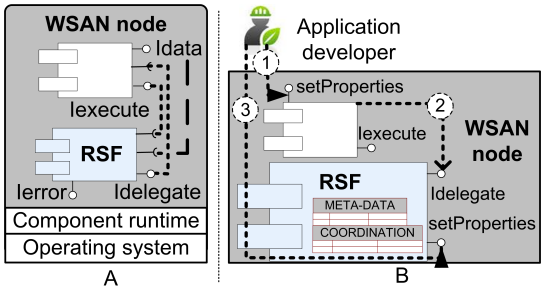

Figure 3: (A) RSF node view, (B) Application assembly

Step 1: Configure component The command used is:
`setProperties(Set-ID, interaction, interval, duration,` $ApC_n[\,], P_n[\,]$`)`
The submitted data is: (i) Set-ID is a unique meta-data set identifier, (ii) interaction type: 1 for periodic or 2 for stochastic, (iii) interval, in the case of periodic interactions it specifies the sample interval for computation. For stochastic interactions, it specifies the type of occurrence that will trigger computation, (iv) duration specifies for how long a configuration is valid, (v) application constraints, (vi) Parameters[], which inform computation (e.g. high or low pass filtering). For example:
`setProperties(001,1,7 min.,31 days, [5%,0,.5, 10%], 0)` This commands configures the temp component for composition 001 (i.e. safety app), in a periodic interaction with a sample interval of 7 minutes, during the next 31 days. The RSF should allow for a time-drift of 5%, and check accuracy using an alpha of .5 and a delta of 10%.
Step 2: Delegate configuration and meta-data Each submission triggers the component instance to delegate all the meta-data to the RSF. This data is persisted in the RSF as a new record in the meta-data table. Each record represents a software composition an instance is part of. Each instance can delegate multiple meta-data sets, one for each application where it is being used in. The RSF then checks for compliance with performance restrictions and application constraints (see Alg. 1). First, it establishes if a response-time constraint has been submitted. Then the RSF checks two things. First, the submitted configuration against performance restrictions. Second, compliance of the response-time constraint. This ensures that erroneous data will not be generated from invoking components in an unready state or non-compliance to app. constraints.
Step 3: Configure composition structure The command used is:
`setProperties(set-ID, thisInstanceID, otherInstance, otherAddress, interfaceType, interactionType)`
This data becomes another record in the coordination table. The data is structured as follows: (i) Set-ID: associates this data to the component instance configuration, (ii) thisIntanceID: the instance in question, (iii) otherInstance: in the case of pull interaction, the other instance is the data-producer and in push interaction, it is the data-consumer, (iii) otherAddress: network address of the node where the other instance is instantiated, (iv) type of the interfaces to connect and (v) interactionType: pull or push. Note that this submission is made directly to the RSF, in a traditional assembly, this would have been submitted directly to the component runtime. For example:
`setProperties(001, temp, filter, IP2, Itemp, push).`
This command connects the `Itemp` interface from the temp to the filter component for composition 001 (i.e. safety app). The filter is running in node with address IP2 and a push interaction is used.

Algorithm 1 Initial configuration check

Meta-data is delegated $\rightarrow$ check R_j,ApC_2
Establish if there is a response-time constraint.$\{ApC_2 \neq 0\}$
for each restriction **do** $\{\forall i[1..n]$ and $\forall j[1..t]\}$
  Check every parameter. $\{check(P_i, R_j)\}$
for response-time **do**
  Check that interval meets response time.$\{interval \leq ApC_2\}$

4.3 Runtime

Composition-Swapping is comprised of three primary steps: composition triggered, composition verified and composition swapped. We describe each of them in Sections 4.3.1 to 4.3.3 respectively.

4.3.1 *Composition triggered*

As execution is signaled, either by the expiration of a timer (Step 1a in Fig. 4) or a context occurrence (step 1b in Fig. 4), the corresponding composition is selected and its configuration is verified on-the-fly. As we follow a reactive on-the-fly approach, there is no need to statistically model each possible interaction (as is needed in preemptive scheduling). E.g. In our deployment RFID cards are detected following a normal bell shaped distribution, and employees walking in an office, follow a Poisson distribution.

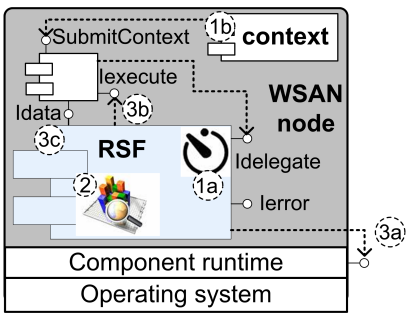

Figure 4: Application integrity is ensured during Composition-Swapping

4.3.2 *Composition verified*

When a composition is triggered, its meta-data is retrieved and passed to the runtime configuration verification process (see Alg. 2). If all required component instances are available and mutex locks are free, computation is scheduled and application constraints are checked ($\forall i[1..RA]$, $(min\{ApC_i\} \leq t_{allowed} \leq max\{ApC_i\})$). If

constraints are met the Composition-Swap is authorized. If the constraints are not satisfied, a *composition-drop* is executed. Where the contending composition is no longer considered for a Composition-Swap and an error event is generated (using `Ierror`). This composition will be reconsidered when it is triggered for execution again. This verification ensures that erroneous data is not generated from the invocation of a sensor or peripheral resource in an unready state or due to violations of application constraints.

Algorithm 2 Runtime configuration verification

```
Composition is triggered → retrieve meta-Data.
if instances are avail. and mutexes are free then
    Schedule execution. {t_expected = t_now}
    if time-drift ApC_1 is respected then
        Authorize Composition swap.
    else
        Composition-drop.
else
    Invoke contention management strategy.
```

In the cases where an instance is busy, the contention management strategy is used. The RSF calculates the time when these instances will be released based on the time each instance started execution and the total time arising from its performance restrictions ($t_{release} = |R_i - (|t_{now} - t_{start}|)|$).

Sensing components: If allowable time-drift specified allows for the latest reading to be reused ($t_{release} \leq A_{td}$). Then a copy is provided for the contending composition (by invoking `Iexecute.copy()`). This not only resolves contention but also reduces energy expenditure from additional invocation of sensors [15, 16]. If not, a *composition-drop* is executed.

Data processing components: The contending composition is queued if its constraints will still be satisfied even after waiting for the release of busy instances ($\forall i[1..RA]$, $(min\{ApC_i\} \leq t_{allowed} \leq max\{ApC_i\})$). If its constraints are violated, a *composition-drop* is done. Delving further into contention management strategies is outside the scope of this paper. The reader is directed to [8], where we discussed alternative strategies, as SLA based prioritization.

Algorithm 3 Contention management strategy

```
for each busy instance i do
    Calculate release time. {t_release = t_release + |R_i −
    (|t_now − t_start_i|)|}
Schedule execution. {t_expected = t_now + t_release}
if instance is a sensing component and time-drift is respected
then
    Request a copy of latest reading.
else if instance is a processing component and time-drift is re-
spected then
    Queue composition to execute later. {t_expected}
else
    Composition-drop.
```

4.3.3 Composition swapped

The swap is comprised of three steps. First, the RSF removes all previous interconnections to the affected component instances and then submits the coordination information to the component runtime in order to assemble the expected composition (Step 3a in Fig. 4). Second, the RSF completes the Composition-Swap by configuring each component, applying corresponding mutual exclusion (mutex) locks and finally signaling the instance for execution (Step 3b in Fig. 4). Third, the RSF monitors the execution of the composition and removes the corresponding mutex locks as each component's delay time expires. In the cases where an accuracy constraint is present, the RSF monitors component output to check data accuracy. This is achieved by connecting the RSF to `Idata` of the component being monitored (Step 3c in Fig. 4). In this way, every time the component produces data, the RSF, receives a copy, which it then uses to perform an accuracy check (see Alg. 4). `Idata` represents a standard interface used to expose the component's functionality. `Ierror` is used to notify of any violations to accuracy constraints. The Composition-Swapping process occurs every time a composition is triggered for execution in the RSF.

Algorithm 4 Accuracy verification strategy

```
Data value is received → retrieve accuracy constraint ApC_3.
Set alpha and delta based on the submitted constraints
if outputType = 1 then
    Run simple exponential smoothing
    if data value is NOT within acceptable delta then
        raise corresponding accuracy constraint violation
else if outputType = 2 then
    Run double exponential smoothing
    if data value is NOT within acceptable delta then
        raise corresponding accuracy constraint violation
```

4.4 Modifiability

Although we consider a performance restriction expressed as delay, and quality of data based application constraints, generic enough to be broadly applicable, we did foresee their extension or modification. Their definitions and corresponding verification processes (see Sec. 4.2.1 and 4.3.2), are isolated and contained in pre-defined modifiable locations. As these locations have been designed as adaptability points in the framework, the scope of change is limited to the interpretation of each restriction or constraint and the check function used. We refer the reader to [8] for further details.

5. EVALUATION ENVIRONMENT AND RESULTS

We now describe the environment used for evaluation of the RSF and its results. We present our evaluation in two parts: reconfiguration efficiency and runtime resource consumption.

5.1 Evaluation environment

Our smart office has been running for over one year, during which time five reconfiguration scenarios were enacted to deploy/extended four distinct concurrently running sensing applications (described in Sec. 2). Each reconfiguration case is comprised of the deployment of a new application. The exception lies in cases one and two, which comprise the initial deployment and extension of the HVAC application. The infrastructure consists of two different platforms: 25 AVR raven nodes [1] and 2 Alix gateways. The raven nodes, have a 20 Mhz CPU, 16Kb of RAM and 128Kb of FLASH memory. They run Contiki 2.5 OS (consumes 40Kb FLASH and 5Kb RAM) [22] and the LooCI middleware (consumes 25Kb FLASH and 5Kb RAM) [11], both programmed with C. Contiki provides support for concurrency, networking and dynamic loading of code, and LooCI provides the component runtime and handles distribution concerns. Alixes have 500 MHz CPUs, Linux and OSGi [19].

5.1.1 The benchmark configuration

The smart office design and benchmark implementation was initially completed using a standard component model, i.e. LooCI. Detailed logs and records were kept as applications were deployed and reconfigured on the 25 Raven embedded nodes [1], keeping track of disruption, developer effort, and resource consumption. All reconfiguration actions, i.e. commands issued to the LooCI API, as well as all events generated by the system and runtime performance metrics for each node were logged. Performance metrics include: dynamic use of stack, heap, CPU[3], and MMEM[4] memory. Access to the smart office's code, configuration files and architectural views used for the benchmark implementation and RSF evaluation can be found at the RSF website (https://code.google.com/p/r-s-f/wiki/smartOffice) .

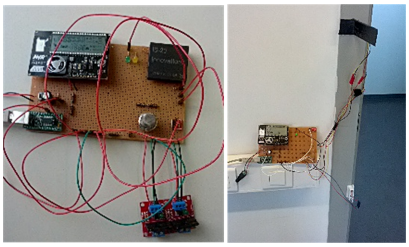

Figure 5: Raven nodes equipped w RFID, CH4, actuators, etc.

5.1.2 RSF implementation and evaluation procedure

We replicated the benchmark configuration using the RSF. The applications deployed for evaluation are functionally identical. We re-enacted all of the reconfiguration scenarios undergone by the benchmark deployment: Case 1, temperature control for the HVAC application is deployed. Case 2, HVAC application is extended with illumination control. Case 3, workplace safety application is deployed. Case 4, comfort application is deployed. Case 5, security application is deployed. We have recorded: disruption, developer effort, resource consumption, reconfiguration actions and node performance metrics just as we did during the benchmark.

5.2 Reconfiguration efficiency

Reconfiguration efficiency includes developer effort, disruption and data transmission experienced during reconfiguration. Developer effort is quantified in terms of the number of decisions required and commands issued during reconfiguration activities.

5.2.1 Quantifying each evaluation metric

All reconfiguration processes (containing over 1,000 actions), commands issued and time elapsed, where recorded and averaged. We quantified disruption in terms of down time, due to interrupted data-flow, experienced by each application during reconfiguration. Additionally a set of experiments where prepared in order to further refine these averaged results, from which we derived a set of equations that describe the observed behavior. The number of decisions required is directly dependent on the possible Configuration Conflicts CC which need to be identified by considering: for each S shared instance, if any of the configuration parameters P and any of the performance restrictions R have conflicts with any application constraint ApC from the RA Running Applications. $CC_S = \sum_{l=1}^{S}(\sum_{q=1}^{RA} ApC_q(R_l + P_l))$, thus the number of Decisions D required to reconcile conflicts in the simplest case is $D = CC_S$. In cases where the entire component graph needs to be understood, e.g. case 4, one must transverse and check all potential configuration conflicts CC for the entire affected component graph, not only the shared component instances.

[3]CPU measurements with Contiki's energest module.

[4]Contiki's managed dynamic memory module.

5.2.2 Developer effort and disruption

As can be seen in Fig. 6, there is a considerable decrease in developer effort and disruption with the use of the RSF after the 5 reconfiguration scenarios. The # of decisions required decreased by over 79%, and the # of commands needed decreased by over 24%. Zero disruption was experienced with the RSF. While with plain LooCI, applications experienced 38 minutes of downtime.

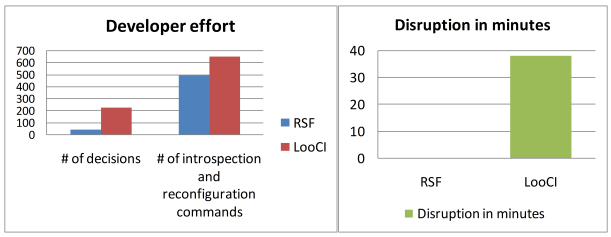

Figure 6: RSF decreases effort and disruption.

RSF case:	1	2	3	4	5	Total
# decisions	28	4	5	5	4	46
# commands	156	96	107	107	29	495
disruption(minutes)	0	0	0	0	0	0
ave. deg. sharing	1	2	2	3	1	NA
% shared components	0	25%	21%	39%	31%	NA
LooCI case:	1	2	3	4	5	Total
# decisions	24	20	25	153	4	226
# commands	148	108	131	239	29	655
disruption(minutes)	0	9	11	18	0	38
ave. deg. sharing	1	2	2	3	1	NA
% shared components	0	11%	7%	11%	9%	NA

Table 1: RSF incurs less reconfiguration effort and disruption.

Breakdown of results for each office per case: Table 1 lists the results obtained for developer effort, and disruption in each office. We also list: (i) The averaged degree of sharing, which represent the number of compositions an instance is shared in, and (ii) the % of shared instances.

In Case 1, there is no component sharing, the worst case scenario for the RSF, there is an increase of 14% # decisions, and 5% # commands.

In Case 2, benefits in all dimensions are noticed for the RSF with a reduction of: 80% in decisions, 11% in commands, and 100% in disruption. The RSF is sharing 1 sensing and 1 processing components per node.

In Case 3, bigger benefits occur across the board because the RSF shares 2 sensing and 2 processing components per node.

In Case 4, even greater benefits are obtained because this reconfiguration requires full introspection scope (see Sec. 2).

In Case 5, there are no measurable benefits as there is no sharing on either system. The RSF imposes no overhead for effort in the cases where no sharing is required, once the overhead of the initial deployment has incurred.

5.2.3 Amount of bytes transmitted

We have measured the amount of bytes transmitted in each office per case during reconfiguration activities (see Fig. 7). The

RSF achieves a reduction of over 33% (i.e. 87Kb) on the averaged amount of bytes transmitted. This is primarily due to component instance sharing in RSF, as no redundant components need to be deployed.

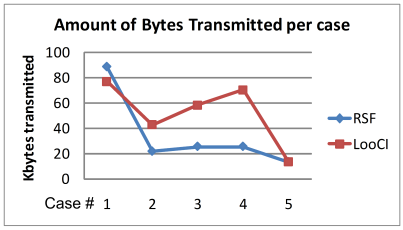

Figure 7: Fewer bytes transmitted with the RSF

5.3 Runtime resource consumption

Runtime resource consumption is quantified in terms of: footprint, performance, processing overhead of the RSF, and the number of nodes required per office.

5.3.1 *Footprint and performance of the RSF*

We have measured static footprint and evaluated performance, in terms of RAM, and FLASH memory usage at runtime:
Footprint: The RSF consumes about 11Kb of FLASH (17% overhead over OS) and 400b RAM (4% overhead over OS).
Memory: We instrumented our code to provide real-time monitoring traces at 5 min. intervals throughout the deployment. As one can see in Fig. 8A and B we plotted the averaged results per node. RAM overhead is compensated for when 25% of the instances are shared and FLASH overhead is compensated for when 39% of instances are shared. Thereafter maintaining a lower per application overhead in the use of memory.

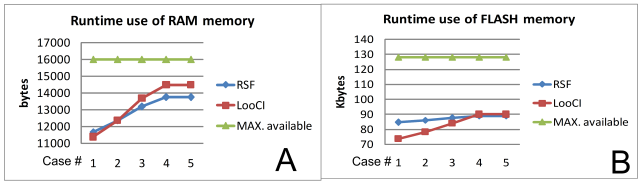

Figure 8: RSF proves resource efficiency.

5.3.2 *Processing overhead of the RSF*

We have measured processing overhead in terms of CPU usage and the time delay created by RSF interactions with the MCU configured at a speed of 8 MHz:
CPU: We have measured CPU usage using the Energest module from Contiki (see Fig. 9). The graph plots averaged CPU usage for all running processes (i.e. Contiki, LooCI, and the RSF) and also illustrates the portion of this CPU usage that is attributable to LooCI and to the RSF. This plot was made based on averaged CPU usage across all nodes at configuration time and runtime. At configuration time, component deployment required 80% of the CPU, for which LooCI accounts for about 50% and the RSF has no noticeable usage. During component instance configuration, 60% of the CPU was used, from which LooCI consumes about 18% and RSF has no noticeable usage. At runtime every time a composition is triggered for execution, on average 55% of the CPU is used, from which LooCI has no noticeable usage and RSF consumes under 5% of the CPU (these are the RSF's peak processes).

Time delay: The RSF imposes time delays due to the exchange of two messages between the RSF and each component (each takes 0.07977 milliseconds) and the time incurred during verification processes (on average 0.159 milliseconds). We have added and averaged these delays taking into account the variations of composition size and complexity of computation. The RSF imposes an averaged delay of .249 ms. A delay which causes no noticeable decrease in the data accuracy or timeliness for any application.

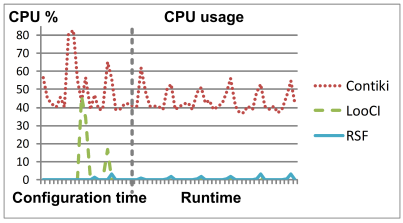

Figure 9: Low CPU overhead with the RSF.

5.3.3 *Fewer nodes required with the RSF*

The RSF requires 30% (i.e. 4) fewer nodes per office. In Fig. 10A, one can see the LooCI component graph, which is functionally equivalent to one of the RSF component graphs shown in Fig. 10B. In Fig. 10B, each graph denotes the triggered composition, during one time period, for one node after case 4 (it takes six time periods to activate all six compositions). The criteria used to modularize functionality is identical for LooCI and the RSF. As a result of composition-swapping, the RSF requires only 10 components to achieve what in a standard model requires 17 components. Furthermore, as each Raven node can support a max. of 12 components, due to limitations intrinsic to the code deployment infrastructure and EEPROM size, more nodes are required to support the same functionality. ID-F implements a string matcher used to identify a user. TrH-F is a threshold filter. TimeF is a down-sampler. AVE is an averager. FAN and LAMP are actuator controls for ventilation and illumination.

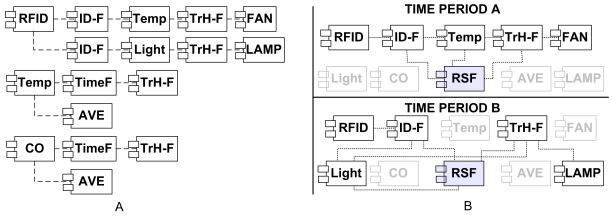

Figure 10: Component graph for (A) LooCI (B) RSF in 2 time periods.

6. DISCUSSION

We have demonstrated two extensions to embedded re-configurable component models, which ensure application integrity and provide resource efficient co-execution. A fundamental primitive in our design was to offer 'add-on' unintrusive extensions. Thus, two key features of the RSF design are particularly notable. First, the RSF itself is implemented as a standard software component and therefore can be integrated with existing models without changes to their supporting middleware. Second, the approach embodied in the RSF can be applied to any re-configurable component model [4–6, 11, 19, 21]. The RSF only requires that component instances are: persistent, have explicit interfaces and are runtime re-configurable. In our prior work [7] we have demonstrated RSF implementations for

various interaction models including: synchronous pull, request-reply interactions using Runes [5], and with asynchronous push, publish-subscribe interactions using LooCI [11].

Achieving lower processing overheads, with comparable features, might have been achieved through the use of more intrusive modifications to the component runtime, OS and system drivers. However, this would lead to cross-cutting code insertions that would have been hard to maintain. Furthermore, making the extensions mandatory, even in the cases when component-sharing is not required by the deployment scenario.

7. CONCLUSION

We have shown that during the deployment of applications on shared environments, the need to *manually* ensure application integrity, causes significant disruption and reconfiguration effort. We identified the two primary limitations that cause this. First, the lack of per-application isolation of component configuration and coordination. Second, the lack of application constraint and component performance meta-data. To address these, we proposed two extensions to re-configurable component models. Mainly, state management delegation and extended component meta-data. These extensions enable component-sharing, which in turn supports Composition - Swapping. Finally, we have shown that for our smart office, Composition - Swapping has: eliminated disruption, lowered reconfiguration overhead and lowered resource consumption. Specifically by reducing, the required decisions by 79%, the commands issued by 24%, and bytes transmitted by 33%. The resource efficiency of our approach can be seen in terms of the lower per-app overhead and fewer nodes required. Its runtime use of RAM is compensated when 25% of the running components are shared. The RSF consumes at most 5% of CPU and its FLASH memory overhead is compensated for when 39% of the components are shared.

Acknowledgment

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IMEC and by the Research Fund KU Leuven.

8. REFERENCES

[1] Atmel, Corp. AVR RZ Raven.

[2] S. Bhattacharya, A. Saifullah, C. Lu, and G. Roman. Multi-application deployment in shared sensor networks based on quality of monitoring. In *IEEE RTAS*, 2010.

[3] R. G. Brown. *Smoothing, forecasting and prediction of discrete time series*. Courier Dover Publications, 2004.

[4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[5] Costa P, Coulson G, Gold R, Lad M, Mascolo C, Mottola L, Picco GP, Sivaharan T, Weerasinghe N, Zachariadis,S. The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In *Percom07*, pages 69–78. IEEE, 2007.

[6] Coulson G, Blair G, Grace P, Taiani F, Joolia A, Lee K, Ueyama J, Sivaharan T. A generic component model for building systems software. In *ACM Transactions on Computer Systems*, volume 26, pages 1–42. ACM, 2008.

[7] P. del Cid, D. Hughes, S. Michiels, and W. Joosen. Applying a metadata level for concurrency in wireless sensor networks. *Concurrency and Computation: Practice and Experience*, 24(16), 2012.

[8] P. J. del Cid, D. Hughes, S. Michiels, and W. Joosen. Evolving wireless sensor network behavior through adaptability points in middleware architectures. *IJDATICS*, 2(1):1–13, 08 2011.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1995.

[10] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen. The gator tech smart house: a programmable pervasive space. *Computer*, 38(3):50 – 60, march 2005.

[11] D. Hughes, K. Thoelen, J. Maerien, N. Matthys, P. J. del Cid, W. Horre, C. Huygens, S. Michiels, and W. Joosen. Looci: the loosely-coupled component infrastructure. In *IEEE NCA12*, 08 2012.

[12] M. Iqbal, M. Handte, S. Wagner, W. Apolinarski, and P. Marron. Enabling energy-efficient context recognition with configuration folding. In *PerCom, 2012 IEEE*, pages 198 –205, 03 2012.

[13] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song. Symphoney: A coordinated sensing flow execution engine for concurrent mobile sensing applications. In *SenSys, 2012 ACM*, pages 211–224, 11 2012.

[14] J. Kramer and J. Magee. Analysing dynamic change in distributed software architectures. *IEEE Software*, 145(5), oct 1998.

[15] I. Leontiadis, C. Efstratiou, C. Mascolo, and J. Crowcroft. Senshare: transforming sensor networks into multi-application sensing infrastructures. In *EWSN*. Springer-Verlag, 2012.

[16] S. Madden and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM TODS*, 30(1), 2005.

[17] Marron P, Lachenman A, Minder D, Hahner J, Sauter R, Rothermel K. TinyCubus a flexible and adaptive framework for sensor networks. In *EWSN05*, pages 278–289. IEEE, 2005.

[18] Mottola L, Picco GP, Sheikh S. FiGaRo: fine-grained software reconfiguration for wireless sensor networks. In *LNCS*, volume 2008, pages 286–304. Springer, 2008.

[19] OSGi Alliance, Org. OSGi - The Dynamic Module System for Java. `http://www.osgi.org/`.

[20] G. Outhred and J. Potter. A model for component composition with sharing. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proc. component-Oriented Programming (WCOP âĂŹ98)*, 1998.

[21] B. Porter, G. Coulson, and U. Roedig. Managing software evolution in large-scale wireless sensor and actuator networks. *ACM Transactions on Sensor Networks*, 9(4):1–28, 2013.

[22] Swedish Institute of Computer Science (SICS). The Contiki OS. `http://www.contiki-os.org/`.

[23] S. Walraven, E. Truyen, and W. Joosen. A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications. In *Middleware*, volume 7049, 2011.

[24] Yu, Y. Rittle, L. Bhandari, V. LeBrun, J. Supporting concurrent applications in wireless sensor networks. In *ACM Sensys*, 2006.

[25] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE06*, pages 371–380. ACM, 2006.